Identifying Website Users by TLS Traffic Analysis: New Attacks and Effective Countermeasures

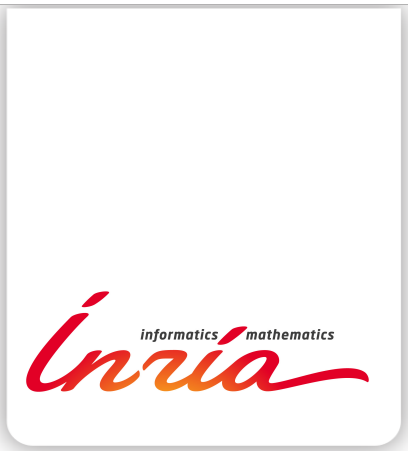
Alfredo Pironti, Pierre-Yves Strub, Karthikeyan Bhargavan

Identifying Website Users by TLS Traffic Analysis: New Attacks and Effective Countermeasures

Alfredo Pironti, Pierre-Yves Strub, Karthikeyan Bhargavan

RESEARCH REPORT
N° 8067
September 2012
Project-Teams Prosecco

ISSN 0249-6399 ISRN INRIA/RR--8067--FR+ENG



Identifying Website Users by TLS Traffic Analysis: New Attacks and Effective Countermeasures

Alfredo Pironti*, Pierre-Yves Strub†, Karthikeyan Bhargavan‡

Project-Teams Prosecco

Research Report n° 8067 — September 2012 — 28 pages

Abstract: Websites commonly use HTTPS to protect their users' private data from network-based attackers. By combining public social network profiles with TLS traffic analysis, we present a new attack that reveals the precise identities of users accessing major websites. As a countermeasure, we propose a novel length-hiding scheme that leverages standard TLS padding to enforce website-specific privacy policies. We present several implementations of this scheme, notably a patch for GnuTLS that offers a rich length-hiding API and an Apache module that uses this API to enforce an anonymity policy for sensitive user files. Our implementations are the first to fully exercise the length-hiding features of TLS and our work uncovers hidden timing assumptions in recent formal proofs of these features. Compared to previous work, we offer the first countermeasure that is standards-based, provably secure, and experimentally effective, yet pragmatic, offering websites a precise trade-off between user privacy and bandwidth efficiency.

Key-words: TLS - Privacy - Anonymity - Web security - Traffic analysis - Protocol implementation - Length-hiding authenticated encryption

* INRIA Paris-Rocquencourt, France - alfredo.pironti@inria.fr
† Microsoft Research-INRIA Joint Centre, France - pierre-yves@strub.nu
‡ INRIA Paris-Rocquencourt, France - karthikeyan.bhargavan@inria.fr

RESEARCH CENTRE
PARIS – ROCQUENCOURT

Domaine de Voluceau, - Rocquencourt
B.P. 105 - 78153 Le Chesnay Cedex

Identification des utilisateurs de services web via une analyse du trafic TLS

Résumé : La vaste majorité des applications web repose sur HTTPS pour protéger, sur le réseau, les données privées de leurs utilisateurs. Nous présentons une nouvelle attaque qui, en combinant les données publiques des réseaux sociaux à une analyse de trafic TLS, permet de révéler l'identité des utilisateurs accédant aux sites web les plus populaires. En réponse à cette attaque, nous proposons un nouveau schéma, qui bien que n'utilisant que des fonctionnalités standards de TLS, permet de dissimuler la taille des données transitant et donc d'aiser l'application des politiques de sécurité. Nous avons intégré notre nouveau schéma de dissimulation de taille à GnuTLS via une API de haut niveau. Nous avons fait usage de cette dernière pour le développement d'un module Apache appliquant notre nouveau schéma à un ensemble de fichiers utilisateurs. Ainsi, nous offrons la première contre-mesure ne reposant que sur des fonctionnalités standards, possédant une preuve de sécurité formelle et efficace en pratique, et qui propose donc un bon compromis entre sécurité des utilisateurs et efficacité en terme de taille des données transitant sur le réseau.

Mots-clés : TLS - Vie privée - Anonymat - Sécurité sur le web - Analyse de trafic - Implémentation de protocols - Chiffrement avec authentification et dissimulation de la taille des données

1 Introduction

When web users connect to insecure networks such as public Wi-Fi, they become vulnerable to network-based attackers who may eavesdrop on their sessions to steal sensitive data, such as credit card numbers, or actively tamper with their sessions, say by redirecting them to a phishing website. To protect their users against such attacks, websites now routinely offer "always on" HTTPS [23], and actively encourage their users to connect via HTTPS to view private data like email and social network profiles[1].

Over HTTPS connections, all messages between the user's browser and the website are encrypted and authenticated using the Transport Layer Security protocol (TLS) [10–12]. For example, on the log in page, the user's username and password are sent encrypted to the website so that the network-based attacker cannot obtain them.

Traffic Analysis. However, in its most common deployment, TLS does not change the *shape* of the traffic: it does not change the order, number, length, or timing of messages. This enables a class of attacks called *traffic analysis* that use statistical analysis of (encrypted) network traffic to uncover private data.

Since TLS operates over TCP, the origin and destination IP addresses of a TLS connection is typically known to the adversary. If the website's location were hidden, using a proxy for example, traffic analysis can still often uncover which website a user is connecting to, based on the time, length, and pattern of communication [16, 20]. Once the website is known, traffic analysis can uncover which pages a user is looking at by analyzing the objects that are downloaded [24], or by the number of clicks and keyboard events required to reach different pages on the website [6].

Identifying Website Users. This paper does not address website fingerprinting. Instead, we focus on *user identifiability*. Suppose that the websites that a user accesses are known, but she always uses HTTPS. Can traffic analysis uncover her identity (username) at these websites?

The answer, for a number of major websites, is yes. We demonstrate attacks on Google, Facebook, Twitter, and Live, that allow a network-based adversary to accurately identify the connecting user, given a target set of users. The key idea is to correlate the TLS traffic generated by a user with the user's public social networking data across these websites. In particular, we show how to identify the public profile photograph of a user as it is downloaded over TLS.

We focus on websites with a strong social networking component. This is because social profiles are highly personalized, yet partially public. Users access their profiles several times a day and connect their social identities with other websites. Users are often logged on to more than one social network at the same time. All these features enable an attacker to amplify the traffic analysis by obtaining many samples of the user's profile data for correlation.

How serious are these attacks? In some political climates, the identities of users on social networks can be sensitive, especially if they could be correlated with other data observed from or stored on the user's machine. Moreover, with the advent of social sign-on, users often log in to a variety of other websites using their social networks. For example, the health website HealthVault allows its users to log in through Facebook and so does the tax preparation website H & R Block. Our attacks apply to HealthVault as well: that is, the attacker can accurately identify which Facebook user has logged in to the website. Combined with other strategies for analyzing the user's behavior on health websites [6], the privacy implications can become quite serious.

On Compression. Our attacks apply to any website where different users' personal data have different but predictable lengths, so that a network-based adversary can distinguish users based on the observed TLS traffic.

Compressing personal data makes the attack even more effective, since two data streams that have equal length but different content are likely to have different (and predictable) compressed lengths. In our attacks, we exploit this property for image compression: even if a website serves equal-sized images for each user, their compressed JPEG length is often distinct.

[1] Gmail started using HTTPS in 2008 and made it the default in 2010. Facebook allows users to choose "always on" HTTPS since 2011.

More generally, compression before encryption is known to leak some bits of plaintext [18]. In the context of HTTPS, both TLS-level compression and HTTP-level compression may be exploited to identify plaintext from a set of known candidates [4]. In particular, this means that HTTP headers (such as cookies) are at risk.

We advocate that automatic compression of sensitive data should be disabled in TLS, HTTP, and in new protocols such as SPDY [17]. When compression is used, it should be carefully reviewed so that the compressed length does not leak too much information about the plaintext. In our proposed countermeasures, we allow application-layer compression but ensure that the length of compressed data is hidden within a large enough range to avoid user identification.

Countermeasures. Our attacks are quite simple; unlike previous studies they do not utilize sophisticated statistical classifiers, and they do not require detailed knowledge of the website's control flow. They rely only on *coarse features* of the network traffic, such as the number of TLS fragments sent by the website and their total length. The obvious countermeasure is to normalize the traffic in a way that different users would become indistinguishable, but doing this efficiently is a challenge.

The TLS protocol offers a per-fragment length-hiding mechanism called *padding*, that enjoys formal proofs of security [22] under certain conditions (see Section 3 for details). Various padding strategies have been proposed to normalize TLS traffic. The main advantage of TLS-level countermeasures is that they can be implemented once and for all for all applications. Their main disadvantage is that application-unaware padding schemes tend to be too inefficient [6], and bandwidth-efficient padding schemes that do not hide coarse features are experimentally ineffective [14].

Application-level countermeasures, such as browser-based obfuscation [21], website-specific traffic padding [6], and traffic morphing [26], have the advantage that they can rely more precisely on the expected traffic and its desired shape. However, they also have disadvantages. First, they are not standards-based and have to rely on modified browsers or web servers. Second, their security is not based on cryptographic guarantees and like all obfuscation techniques they may be defeated by increasingly sophisticated statistical analyses.

Anonymity networks, such as onion routing [13], use proxies to forward application data over TLS. They can be quite effective in hiding which website a user is accessing, but for the problem examined in this paper, where the adversary is on the same network as the user and the website is known, their network-based infrastructure cannot help.

In this paper, we focus on TLS-based solutions that websites can deploy with little or no help from the network or from the browser. We systematically explore the effectiveness and efficiency of TLS length-hiding and evaluate it using a concrete implementation on real-world traffic. Our approach is complementary to and may be used in combination with application-level countermeasures.

A New Length-Hiding TLS Implementation. We design a novel application-aware fragmentation and padding scheme that modifies both the number and length of packets to normalize the network traffic of all application-level messages in a given anonymity set. Our scheme is fully compatible with TLS and can be seen as an elaboration and implementation of length-hiding authenticated encryption as formalized and proved in [22].

We implement this scheme as a patch to several TLS implementations, including GnuTLS (in C), JSSE (in Java), and a custom .NET server (in F#). Each implementation exposes an API that an application may use to request customized length-hiding for its messages. As a concrete example, we implement a configurable Apache module that uses the GnuTLS length-hiding API to hide the lengths of all files in a directory, based on an anonymity policy.

The overhead of our scheme depends on the specificity of the resources the website wishes to protect. To prevent our traffic analysis attacks on profile photos, making each user indistinguishable from any other, the overhead can be as much as 770% for Google Plus profile pictures or as low as 62% for Facebook thumbnail pictures. Nevertheless, if the application cleverly chooses its anonymity range, the Facebook thumbnail picture overhead can drop as low as 15% on average and still give anonymity among 25% of users, which, assuming our experimental results scale to the full set of Facebook users, still includes about 250 million users. Notably, all these numbers are far lower than

those proposed in previous work to prevent web page identification using padding [6, 14].

Relating Implementations to Formal Security Results. Implementing length-hiding TLS is not easy. The only mainstream TLS implementation that attempts to hide lengths is GnuTLS, and it does not even try to modify the number of fragments. [14] shows that GnuTLS's strategy of per-fragment random padding is experimentally ineffective in hiding total bandwidth; per-session random padding does a little better. We show that GnuTLS's strategy of random padding is fundamentally flawed against repeated sampling of user data. In general, any random padding strategy that allows the same plaintext to be sent several times with different paddings is bound to fail. We debunk the assumption that random padding has the same effect as rounding [6]; rounding is much more effective.

How do GnuTLS and our implementation relate to the formal proofs of [22]? We argue that our strategy satisfies the formal hypotheses of [22] whereas GnuTLS does not, despite the latter being the main motivating example for the work in [22].

Implementations often uncover hidden hypotheses in formal proofs. Our work reveals an unstated assumption in the definitions of [22] that manifests itself as a timing attack on mainstream TLS implementations. We show that even if the sender uses length-hiding to obtain same-length ciphertexts from different-length plaintexts, the attacker can distinguish between them by observing decryption time at the receiver. This is because TLS uses an authenticate-then-encrypt scheme, which means that the MAC verification happens over plaintext and leaks the plaintext length in its running time. The leak would not occur in encrypt-then-authenticate scheme. The timing leak is small but measurable. In our implementation, we reduce the leak by writing new MAC and decode functions whose running time depends only on ciphertext (not plaintext) length.

Contributions. Our main contributions are:

- We present a new traffic analysis attack that uses social network profiles to identify users connecting to major websites and demonstrate this attack on real-world traffic.
- We show that the padding strategy used by GnuTLS is ineffective in concealing website users against statistical analysis.
- We present a new length-revealing timing attack against the authenticated encryption scheme of TLS and show that it results in small timing leaks in mainstream TLS implementations.
- We present the first implementation of TLS-level traffic padding that can effectively conceal the identity of website users. In contrast to recent negative results, we show that application-level configuration and TLS-level length-hiding can be combined to obtain an efficient countermeasure.
- We relate our countermeasure and others to recent formal security proofs of length-hiding for TLS. Our implementation uncovers hidden assumptions in the theory.

Outline. Section 2 presents our traffic analysis attacks. Section 3 reviews the length-hiding features of TLS and their formal security properties. Section 4 presents a new timing attack on length-hiding. Section 5 evaluates popular padding and fragmentation strategies. Section 6 presents our length-hiding framework, while section 7 shows how its implementation can be used by websites to efficiently provide different levels of anonymity to its users. Section 8 reviews related work and section 9 concludes.

2 Identifying HTTPS Users by Traffic Analysis

Many major websites now feature a strong social networking component. They encourage their users to create partially public profiles, including personally identifiable data such as photographs, email addresses etc. These

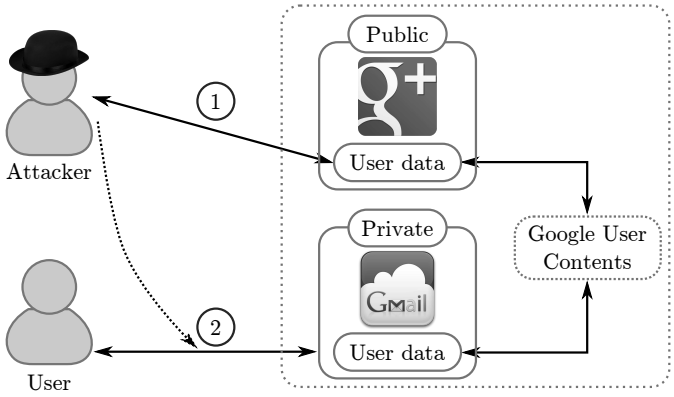


Figure 1: Identifying HTTPS Users using Social Network Profiles

Table 1: URLs of user profiles and pictures on major websites

Website	Public Profile API	HTTPS Image URL
Google	googleapis.com	lh*.googleusercontent.com
Facebook	graph.facebook.com	fbcdn*.akamaihd.net
Twitter	api.twitter.com	twimg*.akamaihd.net
Live	apis.live.net	*.storage.msn.com

profiles are then used as part of the website's directory[2] and linked to profiles of the user's "friends". For example, Google, Facebook, Twitter, and Live, all offer a public directory of subscriber profiles and all of them offer APIs for other websites and applications to programmatically access their content. We show how to use this data to mount a traffic analysis attack.

2.1 Downloading User Profiles

When a user logs in to such websites over HTTPS, her profile data is downloaded, so that she can be given a personalized experience. For example, her name and profile picture is downloaded and displayed on the top of the page. Table 1 shows both the public HTTP URLs and the HTTPS URLs for the profile images served by major websites. Hence, some of the resources downloaded over HTTPS after the user logs in are both user-identifying and also publicly downloadable. This observation leads to our attacks.

We downloaded the public profiles of random sets of users from Google, Facebook, Live, and Twitter; we assume that these sets are representative. Each profile typically consists of some amount of text and a single profile picture. We focus on profile pictures since they are clearly unique for each user and because their length varies significantly across users.

Table 2 describes the dataset that we built gathering profile pictures. The dataset size column tells how many users are in the dataset. If the same profile picture is used in different sizes (e.g. Google Plus home page, and Gmail home page), then the image sizes column report the size for all the gathered images. The third column tells, for each image size, the minimum and maximum byte-length of the images in our dataset, while the last column shows what would be the uncompressed bitmap size (with α-channel) of such images. (When the images size is not fixed

[2]See http://www.facebook.com/directory/people/

Table 2: Dataset of user-linked profile pictures at popular websites.

Website	Dataset size	Image sizes (pixels)	Length range (bytes)	Raw size (RGBA, bytes)
Google	931	27x27, 250x250	[620–2369], [1689–138609]	2916, 250000
Facebook	522 (200[b])	50x50, 180x180[a]	[823–3742], [978–25288]	10000, 344800[d]
Twitter	238[b]	128x128[a]	[1154–87923]	92160[d]
Live	1039	96x96[ac]	[408–769474]	8 MB[d]

(a) Variable size; median reported. (b) Subset of our Google dataset users that also have such an account.
(c) Animated GIF images allowed. (d) Based on the maximal image size of the dataset.

by the web service, we use the biggest image of our dataset for computing the uncompressed bitmap size. This size represents the minimal size that the service would have to use if moving to a fixed image size policy, while allowing the users to keep their profile pictures.)

Figure 2 shows the image size distribution for each profile picture kind, where the x-axis reports the size in bytes, and the y-axis its probability. For pictures with fixed dimensions (like the Google ones and the Facebook thumb ones), the size distribution is unsurprisingly Gaussian-like, with the high tail being significantly large, due to different compression policies applied by some users to their profile images before uploading them to the social networks.

2.2 Identifying Profile Pictures in TLS Traffic

It is well known that TLS as it is commonly deployed does not hide the length of the exchanged content; so it is possible to trace contents flow by analyzing the bandwidth of the HTTPS traffic [9,14,24]. We exploit this weakness to recognize website users.

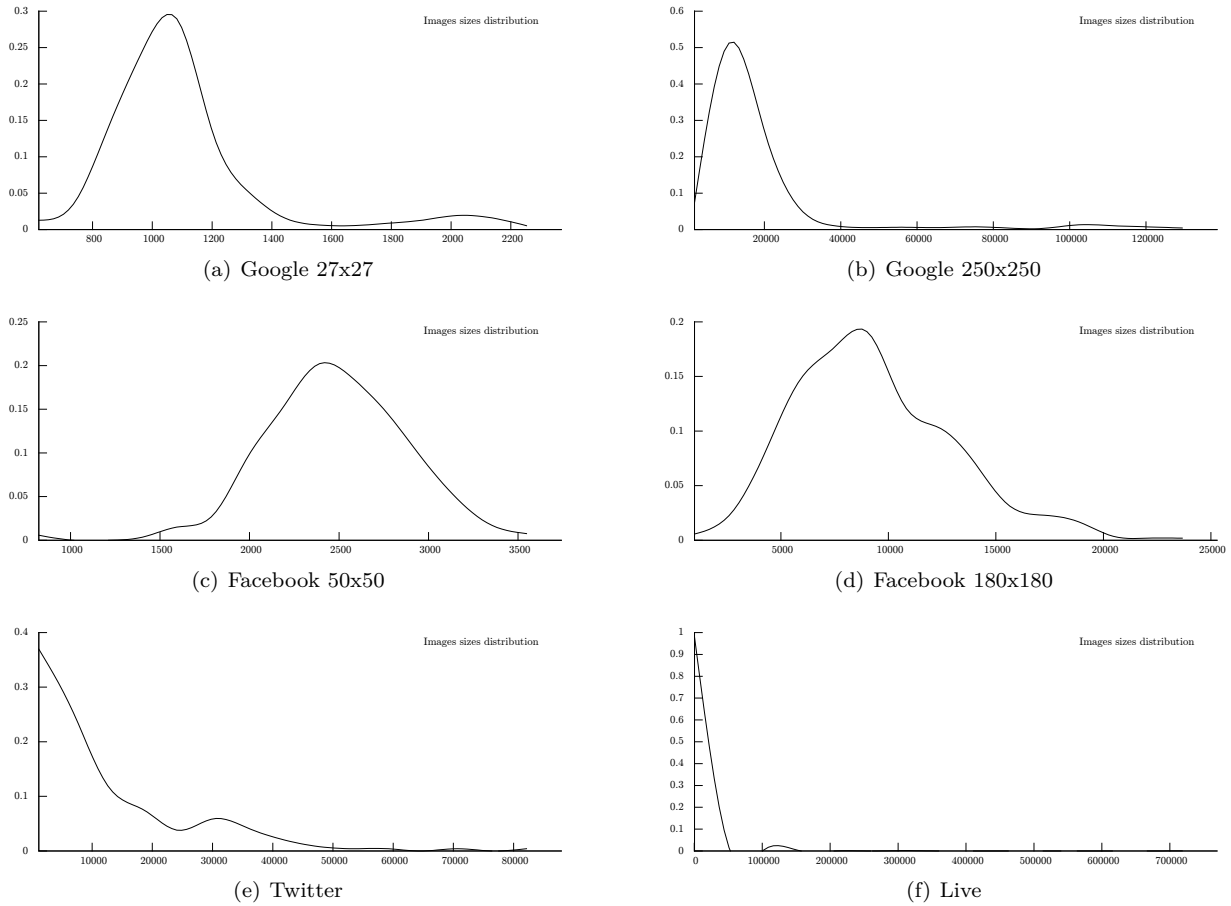
Our attack proceeds as follows. Taking Google as an example, profile pictures are stored on dedicate hosts named `lh[0-9].googleusercontent.com`. The attacker first downloads from the public directory of Google Plus the profile pictures $p_1, \ldots, p_n$ of n users $u_1, \ldots, u_n$ he wishes to recognize (step 1 of figure 1). Given the size in bytes of each picture p_i, the attacker can pre-compute the total number of bytes b_i that the Google server will send when delivering p_i to u_i over HTTPS. Then, when a user logs in to Gmail, the browser makes an HTTPS request to one of the `googleusercontent` hosts, and since server host names are not protected by TLS [15], it is easy for the attacker to identify and track the connection. He then measures the length of the HTTPS response (step 2 of figure 1), and compares it with his pre-computed image lengths to identify the connecting user within a small anonymity set.

Our experiments focus on profile pictures, but logging in to a website normally involves many more data exchanges. For instance, loading the Gmail page generates about 80 HTTPS connections; even under the same initial conditions (e.g. empty caches) the total bandwidth varies significantly at every connection, even for the same user. Several user-identifying resources other than profile pictures are exchanged in these cases (personalized JSON data, XML etc.), and the attacks and countermeasures we discuss here for profile pictures may well apply also to these resources.

2.3 Security Definitions

We define a privacy goal and attack measure; both are closely related to the standard notion of k-anonymity, but are recast to more conveniently state our experimental results.

From the viewpoint of a website, the privacy goal is g-anonymity (or group anonymity):



(a) Google 27x27

(b) Google 250x250

(c) Facebook 50x50

(d) Facebook 180x180

(e) Twitter

(f) Live

Figure 2: Image size distribution.

Table 3: $\frac{m}{n}$-identifiability using profile picture lengths.

	Number of $\frac{m}{n}$-identifiable users											
	Google (27x27, 250x250)				**Facebook (50x50, 180x180)**				**Twitter**		**Live**	
m	$n = 931$				$n = 522$				$n = 238$		$n = 1039$	
1	292	(31.36%)	893	(95.92%)	354	(67.82)	480	(91.95%)	236	(99.16%)	894	(79.19%)
2	232	(24.92%)	38	(4.08%)	128	(24.52)	42	(8.05%)	2	(0.84%)	104	(9.21%)
3	180	(19.33%)	-		24	(4.60)	-		-		21	(1.86%)
4 - 6	227	(24.39%)	-		16	(3.07)	-		-		20	(1.77%)

Table 4: $\frac{m}{n}$-identifiability for Google users by correlating with other data.

$n = 931$ (Google dataset)	**Number of $\frac{m}{n}$-identifiable users**							
m	**+ Facebook**		**+ Twitter**		**+ SNI**		**All combined**	
1	416	(44.68%)	507	(54.46%)	652	(70.03%)	789	(84.75%)
2	250	(26.85%)	184	(19.76%)	198	(21.27%)	112	(12.03%)
3	135	(14.50%)	114	(12.24%)	81	(8.70%)	30	(3.22%)
4 - 6	130	(13.96%)	126	(13.53%)	-		-	

> A website preserves *g-anonymity* if its users are divided into g equal-sized groups, and the identity of each user is indistinguishable from all the other users of her group.

For example, 1-anonymity means that a user is indistinguishable from every other user; whereas with 4-anonymity each user is indistinguishable from one fourth of the user base. We define anonymity this way because it does not depend on the number of users of a system, and can be used to estimate the privacy obtained by a countermeasure for different datasets.

Conversely, from the viewpoint of the attacker, the goal is $\frac{m}{n}$-identifiabilty.

> A user in a target set of n users is $\frac{m}{n}$-*identifiable* if her identity is indistinguishable from $m - 1$ other users in the set.

For instance, a user is $\frac{1}{n}$-identifiable if she can be uniquely identified within the target group, say because her profile picture size is different from everyone else. A user is $\frac{3}{n}$-identifiable if there are exactly 2 other users in the target group from whom she is indistinguishable.

We use the number of $\frac{m}{n}$-identifiable users in a dataset to estimate the success rate of an attack. Intuitively, if n users are using a Wi-Fi network that the attacker is monitoring, then the attacker is able to narrow down the identity of each $\frac{m}{n}$-identifiable user among m potential usernames. The goal of the attacker is $\frac{1}{n}$ identifiability, but as n increases, he may have to be content with larger values of m.

2.4 Experimental Results and Attack Amplification

We compute $\frac{m}{n}$-identifiability for users in the datasets of Table 2, and Table 3 summarizes the results: for each service, the number of users belonging to each $\frac{m}{n}$-identifiability set is reported. Focusing on Google, we can uniquely identify 30% of the users in our dataset when they download the small version of their profile picture (27×27 as

used by Gmail). Nearly half of the users have $m \leq 3$. Facebook gives us a higher attack success rate. Indeed, more than 65% of the users can be uniquely recognized, and less than 10% of users have $m > 2$.

We can greatly increase the attack success rate by cross-referencing data from different social networks, or using auxiliary identifiable information. Suppose we have in our dataset the Google and Facebook profile pictures for the same user, of sizes s_G and s_F respectively. The main idea is that, even if the user is not uniquely identifiable by her Google or Facebook picture sizes, maybe she is the unique user in our dataset to have an s_G picture size for Google, *and* an s_F picture size for Facebook. So, assuming the requests coming from the same client belong to the same user, by observing the picture sizes of different social networks we can better identify and correlate users.

In practice, Google Plus users can link their Google profiles with their profiles on other social networks. We made use of this feature and, for our set of Google users, we downloaded their related Facebook and Twitter profiles. About 20% (resp. 25%) of the Google Plus profiles in our dataset has been linked to Facebook (resp. Twitter) profiles. The first two columns of Table 4 give the $\frac{m}{n}$-identifiabilty rates for users downloading both their thumb Gmail and Facebook (resp. Twitter) profile pictures. In that case, 55% of users accessing Gmail and Twitter have $m = 1$, with 25% of users having $m > 2$, which is much better that the results in Table 3.

The attack success rate can be further increased by exploiting the TLS *Server Name Indication* (SNI) extension [15]. TLS SNI allows a client to indicate the server which hostname it attempts to connect to. This information appears in clear text form, allowing the server to do hostname based dispatch before the handshake phase, when doing HTTP virtual hosting for instance. As state above, Google spreads the profiles pictures among 10 different hostnames, namely `lh[0-9].googleusercontent.com`. From our experience, we noticed that, for a fixed profile picture and geographic location, Google constantly gives the same hostname for all profile pictures of the same user. So, for each user of our Google dataset, we collected the hostname associated to the profile picture. Third column of table 4 shows $\frac{m}{n}$-identifiability when cross-referencing TLS SNI information with user profile picture sizes. In this case, 70% of users have $m = 1$. Moreover, for all users we have $m \leq 3$, instead of $m \leq 6$ of previous cases.

Cross-referencing all this information (Gmail, Twitter and Facebook profile pictures with TLS SNI), nearly 85% of the users have $m = 1$. Only 3% of users have $m = 3$, with the remaining 12% having $m = 2$.

2.5 Countermeasures

Can our user-identification attacks be prevented by simple website countermeasures? Not effectively and not without substantial cost. What if all profile images were replaced by constant-length bitmaps? As Table 2 shows, bitmaps are more than 10 times the size of the compressed images, so this would entail a significant bandwidth overhead. What if the profile data were obfuscated? Application-level countermeasures such as obfuscation [21] may help but require browser-support and are not provably secure.

Our goal is to help privacy-conscious websites protect the privacy of their users. In the rest of this paper, we investigate whether the TLS protocol itself can be used to effectively and efficiently conceal user identitities.

3 Length-Hiding in TLS

TLS aims "to provide privacy and data integrity between two communicating applications." [12]. It consists of primarily two sub-protocols: Handshake and Record.

The Handshake protocol reliably negotiates cryptographic parameters such as ciphersuites, establishes shared keys, and authenticates the server identity, typically by a public-key certificate. Since the initial handshake messages of each TLS connection are sent in the clear, a network-based attacker can learn the protocol version, the ciphersuite to be used, and the identity of the server.

The Record protocol provides private and reliable connections that participants may use to send a stream of *fragments* to each other. The protocol uses shared key authenticated encryption, and supports three kind of

Table 5: MAC-then-Encode-then-Encrypt Scheme for TLS

$\mathsf{Enc}_K(c, H, M)$	$\mathsf{Dec}_K(H, C)$
$(K_{ma}, K_{se}) \leftarrow \mathsf{derive}(K)$	$(K_{ma}, K_{se}) \leftarrow \mathsf{derive}(K)$
$T \leftarrow \mathsf{MAC}_{K_{ma}}(H, M)$	$X \leftarrow \mathsf{sdec}_{K_{se}}(C)$
$X \leftarrow \mathsf{encode}(c, M, T)$	$(M, T) \leftarrow \mathsf{decode}(X)$
if $X = \perp$ return $\perp$	if $M = \perp$ return $\perp$
return $\mathsf{senc}_{K_{se}}(X)$	if $T \neq \mathsf{MAC}_{K_{ma}}(H, M)$ return $\perp$
	return M

ciphers [12]: stream ciphers, such as RC4; block ciphers, such as AES; and authenticated encryption with additional data (AEAD) ciphers, such as GCM. With stream and block ciphers, the protocol employs a MAC-then-Encode-then-Encrypt (MEE) scheme to provide authenticated encryption for each fragment.

The keys and algorithms for the MAC and encryption are established during Handshake: the keys are derived from the shared secret, and the algorithms are specified by the negotiated ciphersuite. For example, the recommended (mandatory) ciphersuite in TLS 1.2 is TLS_RSA_WITH_AES_128_CBC_SHA, which means that encryption uses the block cipher AES in cipher-block chaining (CBC) mode with 128-bit keys, and MACs are computed with HMAC-SHA1. (RSA is used only in the Handshake.) As another example, the most commonly-used ciphersuite on the web is TLS_RSA_WITH_RC4_128_SHA, which uses the stream cipher RC4 for encryption.

With stream and AEAD ciphers, the length of the plaintext can be calculated from the length of the ciphertext and the chosen ciphersuite, both of which are known to a network-based attacker. That is, stream and AEAD ciphers do not provide any mechanism to hide the length of the plaintext and thus are of no use to defeat our attack. Conversely, block ciphers require the plaintext to be *padded* so that it is a multiple of the block size, and this can be used to hide the real length of the plaintext. In the rest of the section, we describe the MEE scheme for block ciphers and its formal length-hiding properties as recently established by [22]. Our presentation closely follows the defintions of [22] to make it easier to relate our work to their formal results.

3.1 MAC-then-Encode-then-Encrypt (MEE)

The MEE algorithm works as follows. For each TLS fragment, it first computes a message authentication code (MAC). It then encodes the fragment and the MAC into a bitstring suitable for encryption. Finally it applies a symmetric encryption algorithm on the encoded bitstring to compute a ciphertext that may be sent over the network.

For block ciphers, the encoding algorithm concatenates the fragment and its MAC and then adds some padding bytes, such that the result is block aligned.

Following [22], the generic MEE scheme, parameterized by algorithms for MAC, encoding, and encryption can be written as an authenticated encryption algorithm as shown in Table 5.

- $\mathsf{Enc}_K(c, H, M)$ takes a message M, a "header" H (consisting of additional data that needs to be authenticated), a target ciphertext length c, and a key K, and returns a ciphertext of length c; and
- $\mathsf{Dec}_K(H, C)$ takes a ciphertext C, a header H, and key K, verifies C, and returns a plaintext.

It is now well known that particularly when a block cipher is used, particular care must be taken in Dec not to allow an attacker to distinguish between a decoding failure and a MAC failure, otherwise plaintext recovery attacks become possible [5, 25]. In TLS-like protocols, even a small timing difference between these two error conditions can sometimes be amplified and exploited [1].

Proofs of MEE (e.g. [22]) rely on *uniform error reporting*, that is, that the TLS implementation ensures that neither the error message nor its timing reveals which of the two errors occurred. In practice, TLS implementations take care to hide the time difference in Dec by computing the MAC whether decoding succeeds or fails. Hence, they typically implement a modified Dec as follows, where t is the size of the MAC. Note the if decode fails, then X is split into M and T anyway, and the failure is postponed. Also, array equality checks (such as the MAC check) are carefully implemented so that they do not return as soon as the first byte in the arrays differ, because this would leak, through timing, the position of the different byte. Instead, array equality always loops through the whole array, and an internal flag holds the result of the computation.

We use the notation $X[i..j]$ to represent the sub-array of X with all indexes from i inclusive to j exclusive. Array indexes start from 0.

$\mathsf{Dec}_K(H, C)$

$(K_{ma}, K_{se}) \leftarrow \mathsf{derive}(K)$
$X \leftarrow \mathsf{sdec}_{K_{se}}(C)$
if $|X| \leq t$ return $\perp$
$(M, T) \leftarrow \mathsf{decode}(X)$
fail $\leftarrow$ false
if $(M, T) = (\perp, \perp)$
    fail $\leftarrow$ true
    $M \leftarrow X[0..|X| - t - 1]$
    $T \leftarrow X[|X| - t - 1..|X| - 1]$
if $T \neq \mathsf{MAC}_{K_{ma}}(H, M)$ fail $\leftarrow$ true
if fail = true return $\perp$
return M

The TLS 1.2 standard [12] recommends this solution but confesses that it "leaves a small timing channel, since MAC performance depends to some extent on the size of the data fragment, but it is not believed to be large enough to be exploitable, due to the large block size of existing MACs and the small size of the timing signal." This assessment is not completely right, since [1] shows that such small signals can sometimes be amplified if the attacker is able to accumulate the timing leaks from a number of decryption queries. In practice however, TLS connections are typically torn down at the first error, hence this timing leak is probably too small to practically exploit. Conversely, in TLS variants like DTLS, where implementations do not stop at the first error, timing attacks do become feasible [1].

3.2 Length-Hiding by Padding

When using a block cipher, all versions of TLS require at least 1 byte of padding. At the very least, plaintext must be padded up to the next multiple of the block size, a scheme we call *minimal padding*. However, TLS allows each fragment to be padded with up to 256 bytes, and advises that "lengths longer than necessary might be desirable to frustrate attacks on a protocol based on analysis of the lengths of exchanged messages" **??**. In other words, extra padding provides some length-hiding.

Mainstream TLS implementations use minimal padding when they send data for bandwidth efficiency. However, we find they still accept messages padded up to 256 bytes, and hence are willing to converse with peers who implement extra padding. GnuTLS was the first mainstream implementation to implement extra padding both at the sender and the receiver.

Table 6: TLS Encoding for Block Ciphers with Padding

```
encode(c, M, T)
if c mod n ≠ 0 return ⊥
r ← (c − |M| − t)
if r > p return ⊥
if r < 1 return ⊥
P ← int2byte(r − 1)
X ← M ‖ T ‖ P···r
return X
```

```
decode(X)
if |X| mod n ≠ 0 return (⊥, ⊥)
P ← X[|X| − 1]
r ← byte2int(P) + 1
l ← |X| − r − t
fail ← false
if l < 0 fail ← true
for i = 2 to r do
    P′ ← X[|X| − i]
    if P′ ≠ P fail ← true
M ← X[0..l]
T ← X[l..l + t]
if fail = true return (⊥, ⊥)
return (M, T)
```

Even minimal padding guarantees that two plaintexts whose lengths round up to the same number of blocks will not be distinguishable by an attacker who observes the ciphertext. This hides lengths within a range of one block-size, which for AES is 16 bytes and for 3DES is 8 bytes.

Padding is implemented in the encode and decode algorithms as shown in Table 6. These algorithms are parameterized by the block-size n, maximum padding-length p, and tag size t. These algorithms are taken from [22] except that we explicitly enforce uniform error reporting in decode.

By choosing ciphertext lengths c between $|M|+t+1$ and $|M|+t+256$, an implementation can hide the plaintext length $|M|$ within a range of 255 bytes. More generally, by fragmenting a message M into F fragments, its length can be hidden in an arbitrarily large range of $F * 255$ bytes. This observation is the core of our length-hiding strategy. However, typical implementations of TLS padding do not fully exercise this length-hiding power, and as we show later in section 5, even those that implement fragment-level padding are often either flawed or ineffective.

3.3 Formal Length-Hiding Authenticated Encryption (LHAE)

[22] presents formal proofs of security for the use of MEE in TLS. In particular, it formalizes and proves that MEE as presented above provides length-hiding authenticated encryption with additional data (LHAE). The LHAE security game is defined in terms of the oracles in Table 7.

- LHAE generates a key K, picks a boolean b, and offers two oracles OEnc and ODec to the adversary A. The adversary wins if it is able to guess b.
- OEnc(c, H, M_0, M_1) uses MEE to encrypt both M_0 and M_1, obtaining C_0 and C_1 both with target length c. If both encryptions succeed, then it adds C_b to the set $\mathcal{C}$ and returns it.
- ODec(H, C) accepts a ciphertext C only if it is not in the set $\mathcal{C}$ and $b = 1$, and it returns the MEE decryption.

The main result of [22] is that (under certain assumptions, detailed below) with the MEE scheme of TLS, the advantage of an adversary playing the LHAE game is negligible over an adversary who randomly picks b'.

Informally, this means that any adversary who is able to observe and tamper with encrypted TLS fragments can still not tell the difference between two plaintexts M_0 and M_1 as long as they have the same target ciphertext

Table 7: Length-Hiding Authenticated Encryption (LHAE) Game

LHAE	$\mathsf{OEnc}(c, H, M_0, M_1)$	$\mathsf{ODec}(H, C)$
$K \leftarrow_\$ \mathcal{K}$	$C_0 \leftarrow \mathsf{Enc}_K(c, H, M_0)$	if $b = 1$ and $C \notin \mathcal{C}$
$b \leftarrow_\$ \{0,1\}$	$C_1 \leftarrow \mathsf{Enc}_K(c, H, M_1)$	return $\mathsf{Dec}_K(H, C)$
$b' \leftarrow_\$ A^{\mathsf{Enc},\mathsf{Dec}}$	if $C_0 = \perp$	return $\perp$
return (b = b')	or $C_1 = \perp$ return $\perp$	
	$\mathcal{C} \leftarrow \mathcal{C} \cup \{C_b\}$	
	return C_b	

length c. Of course, not all target lengths are implementable, in particular the lengths of M_0 and M_1 can differ by at most 255 bytes.

The hypotheses made in [22] under which the formal result applies to MEE as used in TLS are

1. The messages M_0, M_1 cannot be too short. More precisely, if M is the shortest message ever sent using MEE, then $|M| + t \geq n$, that is the plaintext plus the MAC must exceed one encryption block. Otherwise, there is a plaintext-recovery attack

2. The messages M_0, M_1 must either both succeed encryption or both fail. As coded in the OEnc oracle, the attacker should not, for example, be able to obtain the encrypted fragment $\mathsf{Enc}_K(c, H, M_0)$ when $\mathsf{Enc}_K(c, H, M_1)$ would fail.

3. The implementation of the algorithm Dec of MEE must implement uniform error reporting.

The first hypothesis reflects a new attack reported in [22]. Suppose M_0 and M_1 are short messages with different lengths (e.g. "Yes" and "No"), and c is large (e.g. 48 bytes). We may then (incorrectly) assume that the two messages are indistinguishable to a network-based (IND-CCA) adversary. However, if the message and the MAC both fit within the first block, such an adversary is indeed able to use specific features of CBC mode to distinguish between the two messages. The problem is that if messages (and MACs) can be that short, then decryption is not *collision-resistant*: two different ciphertexts may both decrypt to the same plaintext. In practice, this may happen with the truncated HMAC extension of TLS [15].

The second hypothesis discounts trivial attacks on the LHAE game as it is set up, and the third hypothesis discounts padding oracle attacks on the MEE implementation.

4 A Length-Revealing Timing Attack on TLS

The main intuition behind the formal result described above is that if the sender sends the same length ciphertext for M_0 and M_1, and if the receiver only accepts validly-created ciphertexts, then the network-based adversary cannot distinguish between the two.

This intuition is false: it relies on a fourth hidden hypothesis about the timing of MEE encryption and decryption.

4.1 Dec as a Length Oracle

If we look carefully at the algorithms for Dec in Table 5 and calculate their running time (for valid ciphertext), we see that although the time taken by derive and sdec is independent of plaintext length, the time for MAC and decode both depend on the plaintext length. The time taken by MAC is proportional to $|H| + |M|$, and the time

taken by decode is proportional to the padding size r, which in case of valid ciphertext is equal to $|X| - l - t$ (see the for-loop in Table 6).

Hence, the total time taken by Dec is closely related to the plaintext size, and yields the adversary a *length oracle*.

Suppose the attacker is trying to distinguish between M_0 and M_1, such that $|M_1| \neq |M_0|$. Even if we use MEE encryption with the same ciphertext length c for both, the adversary can learn which message was encrypted by sending the ciphertext to the recipient and observing the time taken for decryption. Formally, in the LHAE game, the attacker sends a valid ciphertext to the decryption oracle ODec and observes the time taken for the oracle to return $\perp$ (which in this case does not denote an error, but instead denotes a valid decryption of a known ciphertext.)

In practice, this means that for existing mainstream implementations of MEE, an adversary can foil any length-hiding mechanism by simply measuring the time taken for decryption. Even though the timing difference for a single fragment is typically on the order of microseconds, we find that this time difference can be amplified over a stream of fragments, especially when the sender is sending many highly padded fragments to hide the plaintext length. We show experimental results of this timing leak for mainstream TLS implementations, including implementations like GnuTLS that attempt to hide plaintext length.

How did we find a timing attack on length-hiding when [22] provides a formal proof? As usual, the answer is that the proof does not consider implementation details such as timing side-channels, except to warn implementors to avoid padding oracles. We find that timing is useful not only to uncover padding oracles; it can also compromise length-hiding in valid ciphertexts.

4.2 Length-revealing Timing Attacks on TLS Implementations

As discussed above, in many mainstream TLS implementations the time taken to decrypt a single valid message depends on the plaintext length, but it is too small to be measured over the network. However, a length-hiding (LH) mechanism can help in amplifying this time, up to the point that it becomes measurable.

LH mechanisms like BuFLO [14], or the one explained below in this paper, send messages of different sizes d and d' using the same bandwidth b for all messages. However, on the receiving side, the time taken to process those b bytes depends on the plaintext length (d or d'), rather than on b.

To show the timing attack in practice, we retrieve with the `curl` command line tool, linked against OpenSSL, the smallest and the biggest 250x250 pixels images from our Google dataset, from a web server that uses a LH mechanism, and measure the time taken by `curl` to process each image. (In fact, the experiment was carried out by using our own LH scheme implemented in Apache as detailed in Section 6.) For the experiment we prefer a simple client like `curl`, rather than a web browser for example, to minimize the application timing noise, and make sure that the measured time mostly depend on the OpenSSL library, rather than the application. The expected result is that `curl` will return earlier when downloading the smallest image.

To minimize random network delays, we set up the experiment as depicted in figure 3. Every time an image was downloaded, we locally buffered via a tunnel all the length hiding TLS fragments sent by the server until the full image was delivered. Then, our tunnel forwarded all these fragments over the localhost to `curl` in a burst, so that no network delay was introduced when `curl` was processing the received data. After the last valid fragment, the tunnel also injected a malformed one, so that `curl` acknowledged immediately after computation on the whole image with a TLS alert. We then measured the time occurred between the beginning of the burst, and the reception of the alert message from `curl`. Note that the tunnel activity simply requires some buffering ability and network control, and so the attack could be easily implemented, for instance, by a rogue wireless access point.

We downloaded the length-hidden smallest and biggest images 1000 times each: figure 4 shows the probability distribution of the execution time for the processing of such images. With the images differing by 136920 bytes, we amplify the time execution difference up to 0.5 milliseconds. We repeated this experiment with other mainstream

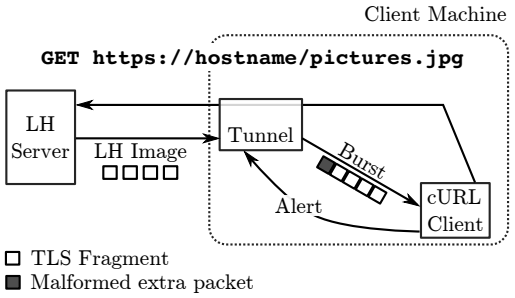


Figure 3: Network setup for the timing attack.

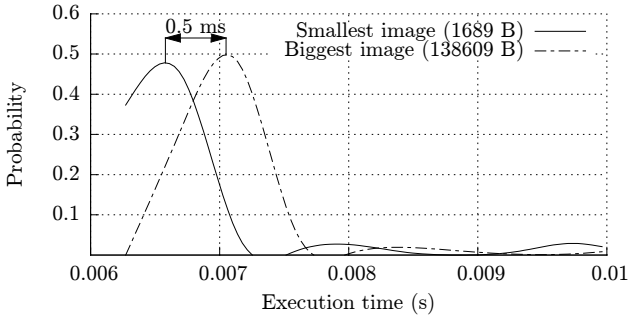


Figure 4: Curl (OpenSSL) execution time for the smallest and biggest Google profile pictures (250x250) transferred with a LH mechanism.

TLS implementations, including GnuTLS, JSSE, NSS, and Bouncy Castle, and obtained a similar time difference.

This time difference is enough to be observed over the Internet [8], and it is comparable with the one measured in [1] for a padding oracle attack based on MAC timing amplification, although the standard deviation measured in [1] was significantly smaller, making the attack easier to exploit.

4.3 Countermeasures: Length-independent MAC and decode

To implement MEE as described in theory, we need to implement MAC and decode in time proportional to the ciphertext length and delink their running time from the plaintext length. Incidentally, this would also remove the "small timing channel" from the padding oracle countermeasure proposed in the TLS standard [12].

Implementing cryptographic algorithms to satisfy specific timing profiles in a provably secure way is an active topic of research [2,3,19] and outside the scope of this paper. We make best-effort attempts for our implementation.

We eliminate the timing leak in decode by modifying the loop to run from 2 to the ciphertext length $|X|$; after the padding has been checked, we perform dummy comparisons for the rest of the loop.

To eliminate the timing leak from MAC, we experiment with two techniques. First, we modify Dec so that it always calls MAC twice, once over the header and plaintext, and the second time over just the pad. The intuition is that sum of the times taken by both calls will be proportional to the ciphertext length. Second, we modify MAC and its underlying hash algorithm so that they take the ciphertext length c as an additional parameter, and we add

extra computations at the end of the hash algorithm to make its time proportional to c. Both techniques seem to help, experimentally, but we leave the full investigation of their timing profile to future work.

More generally, we find that implementing length-hiding in practice requires an incredible amount of discipline. There are a variety of operations on the plaintext that may reveal its length; in some programming languages even the time taken to copy the plaintext between the application and the protocol may leak its length. Of course, the application itself may performs many length-revealing operations on the plaintext.

Our approach is two-fold: at the protocol implementation level, we ensure that all operations depend only on ciphertext-length; at the application-level, we recommend a provably secure timing countermeasure, such as *bucketing* [19], which hides the precise time taken for different operations within time ranges. In other words, we advocate padding for both length and time.

5 Common Fragmentation and Padding Schemes

After having described the MEE encryption algorithm in Section 3 and analyzed some of its timing properties in Section 5, we now describe how existing implementations use the MEE padding features to provide some length hiding mechanisms, and evaluate their effect on our attacks of Section 2.

When a TLS implementation receives application data, it typically first decides how much data to send in each fragment and then how much padding to add to each fragment. The first decision is based on its *fragmentation* scheme and the second is based on its *padding* scheme. The properties of a length-hiding mechanism depend on both schemes.

5.1 Greedy Fragmentation

In most mainstream TLS libraries, the fragmentation scheme is greedy, that is, as much data as possible is sent in each fragment, as soon as it is received from the application. This is considered the most efficient use of the underlying TCP connection.

For our attacks of Section 2, suppose the application delivered to the TLS library the full image file in one chunk as a byte array of length d. The TLS library would consume as many bytes as possible, up to the maximum application data size per fragment FS (typically 2^{14}, but configurable via a TLS extension [15]), resulting in $\lceil d/FS \rceil$ fragments.

In practice however, applications send smaller data chunks to the TLS library, according to some internal buffer size B. For example, a majority of TLS application data fragments sent by Google are 1345 bytes in length (when using RC4 with SHA1), probably because of some file input buffer of size $B = 1325$. When a TLS implementation receives such a small message, it sends it out immediately. Hence, for an image file of size d, the number of fragments would be $\lceil d/B \rceil$.

Once the fragmentation policy is known, the number of fragments is itself a good estimate of the length of the image file. Crucially, no amount of per-fragment padding can help disguise its size. Indeed, for our Google-250 dataset, just the number of fragments sent according to the above fragmentation strategy is enough to distinguish about 2% of the users.

In the remainder of this section, we assume that the fragmentation strategy is unpredictable, and we examine whether per-fragment padding schemes can help hide the precise length of application data sent in each fragment.

5.2 Random Padding - GnuTLS

We begin with the strategy used by GnuTLS, the only mainstream implementation that attempts to use greater-than-minimal padding for length-hiding. Similar strategies have been experimentally evaluated before in several

works [6, 14, 24]. Moreover, GnuTLS is one of the motivating examples for the formal developments of [22], and so it is worth evaluating whether the implementation matches the theory.

Given some data of length d to be sent in a fragment, GnuTLS adds a random amount of padding, between 1 and 256 bytes, such that the resulting plaintext concatenated with the MAC will be block-aligned. The argument is that by adding this random padding, each fragment's true length d is hidden within the range d and $d+P$, where P is conservatively 255, although in practice smaller amount of padding may need to be used to let the final plaintext be block aligned.

However, this argument is flawed if the same application data may be sent several times by the server and be padded with different amounts each time. Indeed, experimental analyses of this strategy [14, 24] show that per-fragment random padding amounts to a small amount of noise that can be easily removed given a sufficient number of samples.

To see why random padding is flawed, consider two messages M_0 and M_1 that have lengths d and $d+k$. Hence, after random padding, M_0 will be between d and $d+P$ bytes whereas M_1 will be between $d+k$ and $d+k+P$ bytes. These two ranges intersect only in the interval $[d+k, d+P]$. So, given sufficient samples, the probability that GnuTLS will pick a padding length that is outside the range becomes overwhelming. For example, if M_0 and M_1 differed in length by 25 bytes, with only 25 samples the adversary will be able to distinguish them with over 99% probability.

In practice, if the sender is willing to send the same message several times with different random pads, then just tracking the minimum ciphertext length gives a good estimate of the plaintext length. Indeed, in our target examples, websites are willing to send the same image many times, either to browsers logged in to the websites or to browsers using social sign-on on other websites. We have experimentally confirmed that adding random per-fragment padding, like in GnuTLS, does not prevent our user-identifying attacks.

How do these length-distinguishing attacks relate to the proofs of [22] which were presumed to cover the GnuTLS strategy? The answer lies in the second hypothesis of the LHAE proof of MEE. The proofs only apply to plaintexts M_0 and M_1 such that either both M_0 and M_1 can be encrypted to fit in the target ciphertext length c or both don't fit.

Per-fragment random padding violates this hypothesis; in particular, it chooses the ciphertext length after choosing the plaintext and may well choose c such that only one of M_0 or M_1 may fit in it. In other words, it implements a different encryption oracle than stated in LHAE, so the proofs do not apply.

5.3 Rounded Padding

We now analyze the padding strategy most commonly proposed as a length-hiding countermeasure, both at the application layer [6, 21] and at the protocol layer [14].

The idea is to round up the plaintext length to the nearest multiple of N. Hence, the plaintext length d is hidden within the range $\lfloor d/N \rfloor * N + 1$ and $\lceil d/N \rceil * N$. The larger the value of N the greater the anonymity set within which the plaintext length may be hidden. Conversely, as N grows, so does the bandwidth overhead, since the sender will now send up to N extra bytes.

At the TLS-level, for each fragment, the maximum value of N is 255. For a stream of F fragments the maximum value of N is $255 * F$. Typical TLS libraries implement minimal padding, that is, they only pad up to the nearest block length. Hence, they use $N = 16$. In that case, assuming a fixed TLS fragment size of 1325 (which improves the efficacy of rounded padding compared to the standard maximum fragment size), we still obtain that 3% of the users of our Google-27 dataset are $\frac{1}{931}$-identifiable, and that 10% of them are at most $\frac{4}{931}$-identifiable, with almost 80% of the users being $\frac{10}{931}$-identifiable at most.

These results are better explained by looking at the image size distribution reported in figure 2. Some images (the ones in the tail of the distribution) will be the only one falling in a specific 16 bytes range, and so they will

still be the unique ones having that rounded padded length. In other words, rounded padding does not ensure that a minimum number of image size will fit in a given slot, and so the attack is still effective, although frustrated.

At the fragment-level, rounded padding schemes do benefit from provably secure length hiding, that is, they satisfy the hypotheses of [22] for all messages M_0, M_1 whose lengths fall within the same multiples N ($\lfloor |M_0|/N \rfloor = \lfloor |M_1|/N \rfloor$). Hence, we debunk the mistaken assumption, for example in [6], that "the effects and the overheads of rounding and random padding are essentially the same, as they both make a packet of a size x indistinguishable from those within the δ-byte range $(\lfloor x \rfloor_\delta, \lceil x \rceil^\delta]$ (rounding) or $[x, x+\delta)$ (random padding)". On the contrary, their effects differ greatly: one is provably resistant to repeated sampling (rounding) whereas the other is not (random).

6 A New Length-Hiding TLS Implementation

Section 5 shows that application-independent strategies such as greedy fragmentation with per-fragment random or rounded padding have limited effectiveness in preventing the attacks of Section 2. In this section, we present a new strategy that combines fragmentation and padding in a single scheme and gives applications fine-grained control on the length-ranges of application data.

We have implemented this strategy for three varied TLS implementations: GnuTLS, an open-source C implementation, JSSE, the default Java implementation of TLS that is distributed with OpenJDK, and a reference .NET TLS implementation written in F#, a dialect of OCaml.

Our implementations consist of three components:

- *CApp* an application-level policy-enforcement module that chooses length ranges (l, h) for application data;
- *CFrag* a generic fragmentation algorithm that takes application data with a desired length range (l, h) and breaks it up into suitably sized fragments with associated length hiding padding $p_1, \ldots, p_n$;
- *CPad* a modified TLS Record implementation that adds p bytes of padding to a given plaintext.

By exploiting both fragmentation and padding, our implementation is able to effectively conceal data lengths in TLS traffic. By using application-level policy enforcement, it allows the application programmer to use her knowledge of the userbase and expected data flow to use tighter, more efficient ranges.

In the remainder of this section, we describe the three components of our GnuTLS-based implementation. The next section evaluates its effectiveness and efficiency.

6.1 *CPad*: Per-Fragment Length Hiding

We implement the Enc and encode algorithms of Section 3 by adding an additional parameter p to the encryption and encoding function. In the case of GnuTLS, this amounts to a patch to the file `gnutls_cipher.c`, where we modify the functions `_gnutls_encrypt` and `compressed_to_ciphertext` that implement Enc and encode respectively. Both functions now take, besides the usual application data length d, an additional parameter p that specifies the length of the hiding pad to be added (still the function will always add the required additional one byte of padding); if the resulting ciphertext is not block aligned (should never happen within our implementation), encryption returns an error.

As described in Section 4, we also modify the MAC and decode functions to run in time proportional to the ciphertext length and not the plaintext length. To implement decode, we modify the function `ciphertext_to_compressed` in `gnutls_cipher.c`. To implement MAC, we modify the underlying Nettle library, specifically the files `hmac-sha1.c` and `sha1.c`.

6.2 *CFrag*: Range-based Fragmentation

Suppose the application wishes to send data of length d but wants to hide it within the length range (l, h). We describe an algorithm that enforces this length-hiding goal by fragmenting this application data into F fragments each with some associated length hiding padding $p_1, \ldots, p_F$. The key feature of the algorithm is that the fragmentation strategy depends only on l and h, not on d. Hence, the number and size of fragments sent will be exactly the same for all application data in the length range (l, h). We present a slightly simplified algorithm than the one we implement, for ease of presentation.

The algorithm RangeSplit below takes a range (l, h) and splits it into two ranges (fl, fh) and (l', h') such that $fl + l' = l$ and $fh + h' = h$. Moreover, it ensures that all plaintexts with lengths up to fh will fit in a fragment. The algorithm relies on the constants FS for the maximum application data size per fragment, PS for the maximum optional pad size, BS for the encryption block-size, and t for the MAC size. For TLS 1.2, with the TLS_RSA_WITH_AES_128_CBC_SHA ciphersuite, these numbers are $FS = 16384$, $PS = 255$, $BS = 16$, $t = 20$.

RangeSplit(l, h)

$fl \leftarrow \mathsf{min}(l, FS)$
$o \leftarrow (fl + PS + t + 1) \bmod BS$
$fh \leftarrow \mathsf{min}(fl + PS - o, h, FS)$
return $((fl, fh), (l - fl, h - fh))$

The algorithm begins by assigning as many bytes as it can to fl, essentially trying to send the minimum l bytes as soon as it can. It then tries to compute the highest possible fh so that it can exploit the maximum length-hiding on offer in the current fragment. fh cannot be higher than h or FS. Moreover, since the maximum number of padding bytes allowed is $PS + 1$, fh cannot be higher than $fl + PS - o$ where o is the remainder that the final size of a fragment made of $fl + PS + t + 1$ bytes would have with respect to the block size. Such remainder would require further (impossible) padding to make it block aligned, and so it is removed. Intuitively, an observer will know that the fragment contains at least fl bytes of data plus some $PS - o$ bytes, but he will be unsure whether the remaining $PS - o$ bytes contain padding or application data.

By calling RangeSplit on a range (l, h), we obtain a range (fl, fh) for the first fragment and a range (l', h') for the remaining data. By calling RangeSplit repeatedly on the remaining range, we obtain a sequence of ranges, until the remainder is $(0, 0)$, which means there is no more data to send.

This sequence of ranges generated by RangeSplit depends only on l and h and consists of two subsequences:

- It begins with a sequence of $\lceil l/FS \rceil$ full fragments each with range (FS, FS), until the l part of the range is exhausted.
- It ends with a sequence of $\lceil h-l/z \rceil$ small fragments each with range $(0, z)$ where $z = PS-(PS+t+1 \bmod BS)$.

(The last fragments of both sub-sequences deviate a little from this pattern in that their ranges depend upon the precise values of l and h.) Hence, for the first l bytes of data, the fragmentation algorithm can send full fragments and is as efficient as possible. The main length-hiding overhead is for the last $h - l$ bytes of data, where each fragment can contain at most PS bytes, since that is the maximum length-hiding on offer per fragment.

Given the RangeSplit algorithm above, we can now define the Fragment algorithm that takes application data M of size $l \leq |M| \leq h$ and splits it into fragments.

```
Fragment(M, (l, h))
(fl, fh), (l', h') ← RangeSplit(l, h)
f ← max(|M| − h', fl)
return (M[0..f], fh − f), (M[f..|M|], (l', h'))
```

RangeSplit returns a fragment range and a remainder range. Hence, Fragment has many choices on how much data to send and how much to keep for the remaining fragments. The algorithm above tries to leave as much data as possible for the last fragment while still sending the minimum fl bytes required by RangeSplit. It returns two outputs: (1) a fragment of the input that is small enough to fit in a fragment and has $p = fh - f$ bytes of length hiding padding, and (2) the remaining byte array and range. Note that, by the definition of fh, the encoded plaintext will be block aligned by padding the returned plaintext by the given p bytes.

Why is it important to keep some data for the end? Since the recipient of the application data may start processing it as soon as he receives it, it is necessary for length-hiding that the recipient only reacts after having received the full application data. For the same reason we also fill the fragments in the reverse order as they are returned by RangeSplit (first the small fragments, then the full ones). Suppose all the data were sent in the first fragment and the rest of the fragments were filled with padding. Then the recipient may respond to the sender even before all the fragments have been received. A network-based attacker who observes this response will then know that the plaintext length was less than h, breaking our length-hiding requirement. In our implementation, we ensure that at least some application data is reserved for the last fragment, but we note that in general, it is difficult to prevent the recipient from reacting to partial data without application support. For example, a web browser may start an HTTP request for a link within an HTML page, before the HTML page if fully loaded.

We implement RangeSplit and Fragment as part of the TLS record implementation. In GnuTLS, this amounts to a modified version of the `gnutls_record_send` function in `gnutls_record.c`. In our implementations, we also try to spread the data more evenly across fragments, to interoperate with TLS implementations such as GnuTLS that drop connections if they receive too many empty fragments.

6.3 *CApp*: Application-level Privacy Policies

Our modified TLS implementation offers a length-hiding API to applications who wish to use it. In particular, it offers a function Send(M, (l, h)) that an application can use to send a message M with range (l, h). Notably, the application must commit to the range, it cannot modify the range later since the fragmentation policy for this bytearray will be based on this range. Of course, once M has been sent, it may send another message M' with a different range.

Using this length-hiding API, applications can directly use our TLS implementations to hide data within any length range. As an example, we show how websites running on the Apache web server may use this API.

We modified the `mod_gnutls` module for Apache to call our length-hiding API. In its basic operation mode, the module now provides a new location directive `GnuTLSRange (<min>, <max>)` which tells Apache to use the LH-API of GnuTLS for all documents served from the given location, using the given range.

For example, suppose all Google user profile pictures of size 250×250 pixels are JPEG files stored in a directory named `/pics_250`, where, according to our dataset, the smallest picture is 1689 bytes and the biggest is 138609 bytes. Then, the following excerpt of the Apache configuration file configures the `mod_gnutls` module to use our length-hiding version of GnuTLS to hide the length of all image files in the directory within the given range.

```
<LocationMatch /pics_250/.*\.jpg>
  GnuTLSRange (1689, 138609)
</LocationMatch>
```

For static files in a physical location, this configuration can be generated automatically and updated as files are modified, added, or deleted. Depending on what the website wants to hide, the configuration may change; for example, to hide the length of HTTP headers as well, their range could be added to the above configuration.

We have also implemented support for g-anonymity in our Apache module by extending the `GnuTLSRange` directive to accept a sequence of g ranges for a set of files:

GnuTLSRange (l0, h0) (l1, h1) ... (ln, hn)

Each file is then hidden within its surrounding range. For files in a directory, these ranges are calculated automatically.

Our Apache module is a simple prototype, and does not yet account for dynamic content, interactive websites, or correlations between content at different locations. Length-hiding support in production websites requires more design and engineering work, but we believe our TLS-based length-hiding implementation offers an important building block for future strategies.

7 Efficient g-Anonymity for Website Users

We evaluate our length-hiding implementation against the datasets of Section 2. By design, websites using our implementation would send the same number of fragments with the exact same fragment sizes for all images in a given directory. As a result, our traffic analysis attacks are rendered completely ineffective. In other words, if profile images were the only user-identifying resource, our implementation would provide full anonymity to all users. In practice, of course, other user-identifying resources would need to be protected as well.

Unsurprisingly, using our implementation produces a bandwidth overhead, since it sends more fragments and more padding bytes to hide the true plaintext length. In the rest of this section, we explore the trade-off between the desired anonymity level and bandwidth efficiency.

Length-hiding Overhead. We begin by estimating the bandwidth overhead of length-hiding. Suppose an application tries to send an image file of size d, and tries to hide it in the range (l, h). The image file is fragmented and then padded.

Each fragment sent over TLS has an overhead O_f that depends on the ciphersuite used. When using a block cipher, O_f consists of:

- 5 bytes for the TLS record header;
- BS bytes for the explicit IV (TLS 1.1 and 1.2);
- t bytes for the MAC;
- 1 byte of required padding

For example, for the mandatory ciphersuite TLS_RSA_WITH_AES_128_CBC_SHA, the overhead per fragment is $O_f = 26$ for TLS 1.0 and $O_f = 42$ for TLS 1.1 and 1.2. Most websites and browsers only support TLS 1.0 (and they accept fragments with non-minimal padding), hence, for direct comparison with other approaches, most of our experiments below use TLS 1.0 and hence the lower overhead number.

The total number of application data bytes plus padding sent by our implementation is h. (For simplicity, we ignore the extra padding needed for block-alignment in the last fragment.) So, the extra padding bytes sent by the implementation is $h - d$. This results in a total overhead of

$$O = O_f * (F - D) + (h - d)$$

Table 8: Range distribution and overhead for profile pictures when using 1- and 2-anonymity

Website	g	grp. size	Thumb images group ranges $[\overline{OH}]$	Big images group ranges $[\overline{OH}]$
Google	1	931	(620, 2369) [1.30]	(1689, 138609) [7.75]
Google	2	466±1	(620, 1113) [0.20]; (1114, 2369) [0.93]	(1689, 18489) [0.54]; (18492, 138609) [4.77]
Facebook	1	523	(823, 3742) [0.62]	(978, 43012) [4.57]
Facebook	2	261±1	(823, 2568) [0.24]; (2572, 3742) [0.35]	(978, 9706) [0.66]; (9709, 43012) [2.69]

where F is the number of fragments sent by our implementation, and D is the number of fragments that would be sent without length-hiding (e.g. using greedy fragmentation). O is in fact a conservative estimation of the total overhead, because it ignores the padding of each of the D fragments when not length-hiding.

As shown in Section 6, for our implementation, the number of fragments is roughly:

$$F = \lceil l/FS \rceil + \lceil (h-l)/PS' \rceil$$

where FS is the maximum application data per fragment (typically 2^{14}) and PS' is the maximum block-aligned padding (251 for the example ciphersuite above).

In comparison, the number of required data fragments relies on the fragmentation policy, that is how many bytes are sent per fragment: $D = \lceil d/B \rceil$.

Ideally, implementations would send $B = FS$ bytes per fragment, but in practice, B may rely on the size of an internal buffer. For example, Google uses $B \leq 1325$ for its fragments.

Given these formulas, we can finally estimate the relative overhead of our implementation as $OH = O/d$, which represents the (fractional) number of additional bytes that are sent for each byte of plaintext.

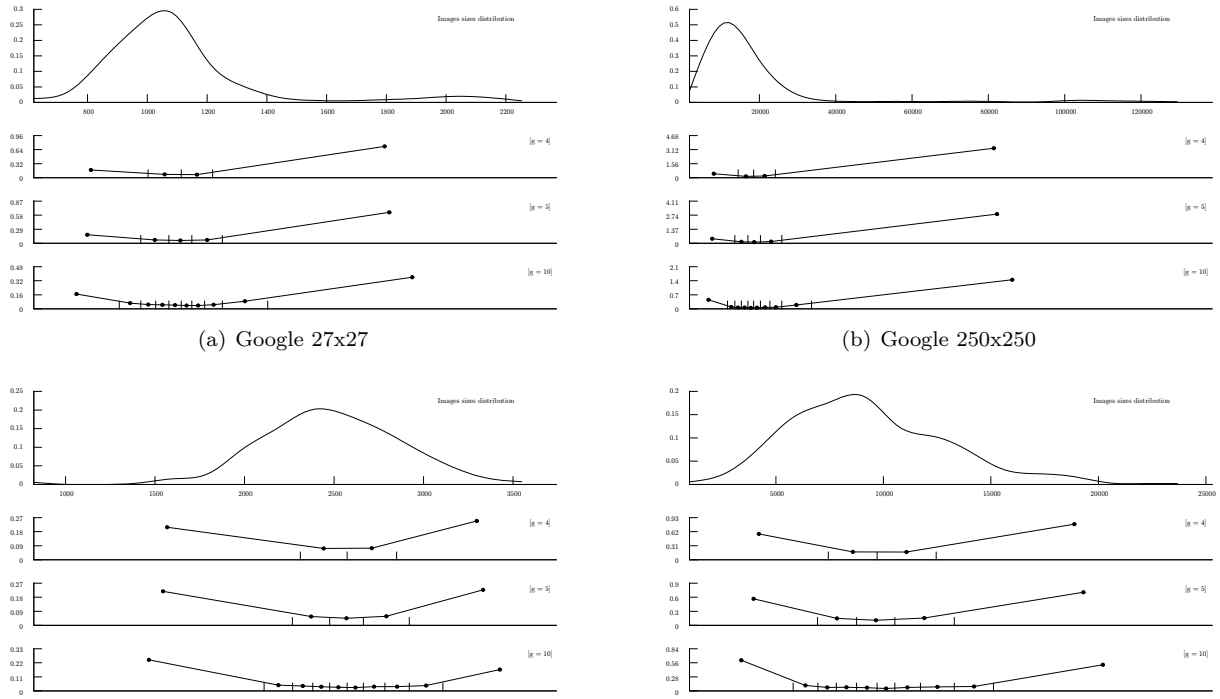
Unsurprisingly, the closer that d is to h and the smaller the interval $h-l$ is, the more efficient our implementation will be. This motivates an application-level strategy to choose ranges that minimize the overhead. In particular, by dividing the set of target application data into multiple equal-sized bins, we obtain lower anonymity but also lower overheads.

Measuring Overhead. We measure the overhead of using our LH implementation on our dataset of Google and Facebook profile pictures, for both the thumbnail sizes (27x27 and 50x50 respectively) and the bigger images (250x250 and 180x180 respectively). In a g-anonymity context, we compute the overhead $\overline{OH}$ of each group as the average of the overhead OH for each image in the group. Table 8 shows the group overheads for 1-anonymity and 2-anonymity. These results are better understood by looking at the profile picture size distribution, as reported in figure 5.

Google distributions, and particularly the one of figure 5(b), are skewed towards the low parts of the graphs. This means that, on average, the picture to be sent will be rather small. This creates significant inefficiency for 1-anonymity, because on average many bytes of pad will be sent. 2-anonymity already improves performances. Indeed, the first group that captures the low half of the users has a smaller range and is rather efficient, while the second group that captures the more spread high part of the users is less efficient. Notably, 2-anonymity is overall more efficient than 1-anonymity, because the second group serves a quasi-uniform distribution, and so average efficiency improves significantly.

Figure 5 also reports a graphical representation for 4- 5- and 10-anonymity. For each g, the x-axis is split according to the range of each group, and the function plots the average overhead $\overline{OH}$ of each group. Not surprisingly, overall efficiency improves, with the groups around the gaussian mean being smaller and more efficient.

In table 9 we measure, on our dataset, the average overhead of using some smaller values of g-anonymity, namely 100 and 50, compared to the bandwidth overhead Google already encounters by using a sub-optimal (1325 bytes) fragmentation strategy. For 27x27 images, which are mostly smaller than 1325 bytes, no adverse bandwidth effect



(a) Google 27x27

(b) Google 250x250

(c) Facebook 50x50

(d) Facebook 180x180

Figure 5: Image size distribution, with range distribution and average range overhead for different values of g-anonymity.

Table 9: Overhead comparison between Google sub-optimal fragmentation policy and 100- and 50-anonymity

Image size	Fragmentation $\overline{OH}$	100-anonymity $\overline{OH}$	50-anonymity $\overline{OH}$
27x27	-	0.04	0.06
250x250	0.02	0.02	0.05

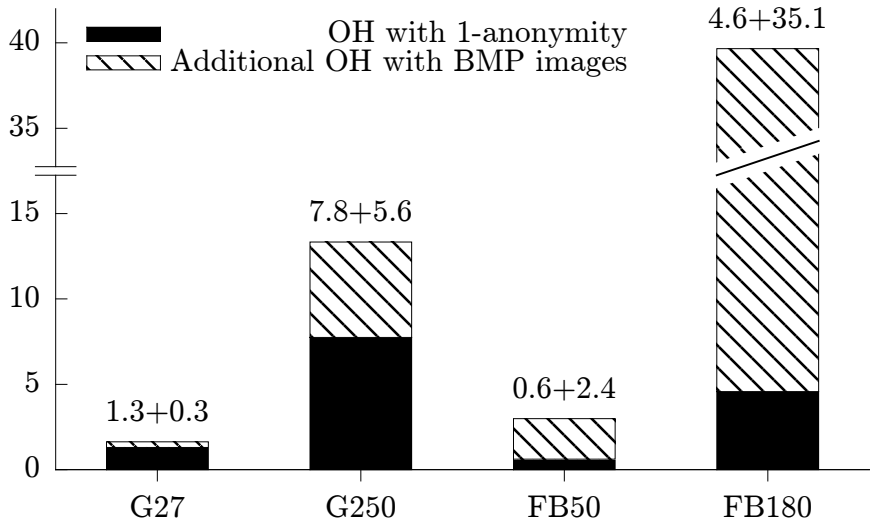


Figure 6: Overheads: 1-anonymity of JPEG versus bitmap images.

is noticeable in practice because of the extra fragmentation. However, for 250x250 images, we point out that the bandwidth overhead due to extra fragmentation is the same as using 100-anonymity with our proposed LH scheme.

How g-anonymity and LH deliver consistent and efficient privacy for all users of the dataset? On one hand, the user-based g-anonymity grouping mechanism ensures that each group contains the same number of users. In contrast, groups generated by size-based mechanisms like rounded padding depend on the size distribution of the profile pictures, crucially creating some groups with a single user inside. On the other hand, having a l value for a (l, h) range enables to be as efficient as non privacy-aware TLS implementations for the first l bytes of payload, whose length is public anyway; the overhead is only paid for the remaining $h-l$ bytes (in terms of extra fragmentation and extra padding). In contrast, previous LH techniques such as BuFLO [14] where not optimizing for public minimal lengths of the payload, increasing inefficiency.

Configuring Website Policies. One might argue that websites could avoid revealing lengths by imposing strict image policies for users. For example, they may only store constant-size bitmap images, rather than compressed JPEGs. Unfortunately, as shown in figure 6, bitmap images tend to be so much larger than JPEGs that our implementation performs better, even for 1-anonymity.

Our implementation is the first that enables websites to carefully tune length ranges to squeeze out the best performance for a given g-anonymity policy. For example, our experiments show that specific values of g perform surprisingly better than others, depending on the dataset. This suggests that websites should profile their web pages to optimize g for different resources, finding a balance between efficiency and privacy. We believe our approach yields a powerful and pragmatic tool for privacy conscious websites.

On the light of the results showed here, websites could also more carefully design their datasets, to provide better privacy. For example, when resizing or recompressing user provided JPEG pictures, they could use a reference target size for the generated file. Reducing the tails of the image size distributions of figure 5 would further improve efficiency of our length-hiding scheme. Size optimization of other user identifying data, such as JSON scripts and dynamically generated content, remains interesting future work; crucially, our length hiding mechanism could be applied similarly.

8 Related Work

Traffic analysis of encrypted web traffic has been a rich field of research with many publications over the last decade [6, 7, 9, 14, 24]. We mention a few relevant publications here.

[24] uses object sizes to identify web pages in real-world traffic. They analyze a large corpus of 100,000 pages to validate their attacks. They also analyze a number of TLS-level and HTTP-level countermeasures for effectiveness against their attacks. Our work finds similar attacks but uses social network profiles to correlate user data. They do not implement any new countermeasures.

[6] presents new attacks to infer user activity on a website. They target sensitive websites such as online tax and online health. They analyze the effectiveness of both rounded padding and random padding and conclude that application-agnostic countermeasures are bound to be inefficient. We show that random padding is inferior to rounded padding and show that rounded padding can be used in combination with application-level anonymity policies to obtain an efficient countermeasure. They propose application level traffic masking techniques; we believe these techniques are complementary to ours.

[14] surveys 9 state of the art TLS-level countermeasures and finds that all of them are ineffective. They suggest that an effective countermeasure must hide the *coarse features* of traffic, such as total connection time, total per-direction bandwidth, and burst bandwidth. We focus on the second one, as we deem it the easier to exploit (but not necessarily to fix) when downloading files. The authors conclude by suggesting: "Future work could investigate more detailed modelings of real-world traffic, and investigate applications of TA countermeasures beyond website fingerprinting. This may uncover settings in which some countermeasures are more successful than they were in our experiments." In response, our work analyzes real-world traffic and focuses on the problem of user identification. We believe that by combining TLS-level countermeasures with application-level policies, we have found an effective solution.

[22] presents the first formal proofs of length-hiding for a MAC-then-Encode-then-Encrypt (MEE) scheme. We show that their results do not apply to random padding and only apply to implementations that do not leak plaintext length in MAC and Decode. Ours is the first TLS implementation that faithfully implements their theoretical definition, and extends it to provide length-hiding over a stream of fragments. Still, formally proving that our implementation itself is secure remains future work.

9 Conclusions

This paper presents new traffic analysis attacks that affect the privacy of users accessing major websites. We demonstrate our attacks on real-world traffic and propose and implement a concrete countermeasure. In contrast to recent negative results, we show that TLS-level length hiding can be effective if combined with application-level policy.

Implementing a length-hiding framework is a major engineering task. Both the protocol implementation and some parts of the application need to be length-aware, since any operation that is dependent on the plaintext length may potentially leak it. Our implementation is a building block that addresses length-hiding; proving it secure or using it to protect applications against other side-channels remain exciting topics for future work.

References

[1] N.J. AlFardan and K.G. Paterson. Plaintext-recovery attacks against datagram TLS. In *Network and Distributed System Security Symposium*, 2012.

[2] Aslan Askarov, Danfeng Zhang, and Andrew C. Myers. Predictive black-box mitigation of timing channels. In *ACM conference on Computer and Communications security*, pages 297–307, 2010.

[3] Daniel J. Bernstein. Cache-timing attacks on AES. Technical report, 2005.

[4] Karthikeyan Bhargavan, Cedric Fournet, Markulf Kohlweiss, Alfredo Pironti, and Pierre-Yves Strub. TLS compression fingerprinting and a privacy-aware API for TLS. In *Provable Privacy Workshop*, 2012.

[5] Brice Canvel, Alain P. Hiltgen, Serge Vaudenay, and Martin Vuagnoux. Password interception in a SSL/TLS channel. In *Advances in Cryptology*, pages 583–599, 2003.

[6] Shuo Chen, Rui Wang, XiaoFeng Wang, and Kehuan Zhang. Side-channel leaks in web applications: A reality today, a challenge tomorrow. In *IEEE Symposium on Security and Privacy*, pages 191–206, 2010.

[7] Heyning Cheng and Ron Avnur. Traffic analysis of SSL encrypted web browsing, 1998.

[8] Scott A. Crosby, Dan S. Wallach, and Rudolf H. Riedi. Opportunities and limits of remote timing attacks. *ACM Transactions on Information and System Security*, 12(3):17:1–17:29, 2009.

[9] George Danezis. Traffic analysis of the HTTP protocol over TLS. Unpublished draft.

[10] T. Dierks and C. Allen. The TLS protocol version 1.0, 1999. Request for Comments 2246, IETF.

[11] T. Dierks and E. Rescorla. The Transport Layer Security (TLS) Protocol Version 1.1, 2006. Request for Comments 4346, IETF.

[12] T. Dierks and E. Rescorla. The Transport Layer Security (TLS) Protocol Version 1.2, 2008. Request for Comments 5246, IETF.

[13] Roger Dingledine, Nick Mathewson, and Paul Syverson. Tor: the second-generation onion router. In *USENIX Security Symposium*, pages 21–21, 2004.

[14] Kevin P. Dyer, Scott E. Coull, Thomas Ristenpart, and Thomas Shrimpton. Peek-a-boo, I still see you: Why efficient traffic analysis countermeasures fail. In *IEEE Symposium on Security and Privacy*, pages 332–346, 2012.

[15] D. Eastlake. Transport Layer Security (TLS) extensions: Extension definitions, 2011. Request for Comments 6066, IETF.

[16] Andrew Hintz. Fingerprinting websites using traffic analysis. In *International conference on Privacy Enhancing Technologies*, pages 171–178, 2003.

[17] Google Inc. SPDY: An experimental protocol for a faster web. http://dev.chromium.org/spdy/spdy-whitepaper, 2012.

[18] John Kelsey. Compression and information leakage of plaintext. In *Fast Software Encryption*, pages 263–276, 2002.

[19] Boris Köpf and Markus Dürmuth. A provably secure and efficient countermeasure against timing attacks. In *Computer Security Foundations*, pages 324–335, 2009.

[20] Marc Liberatore and Brian Neil Levine. Inferring the source of encrypted http connections. In *ACM conference on Computer and Communications Security*, pages 255–263, 2006.

[21] Xiapu Luo, Peng Zhou, Edmond W. W. Chan, Wenke Lee, Rocky K. C. Chang, and Roberto Perdisci. HTTPOS: Sealing information leaks with browser-side obfuscation of encrypted flows. In *Network and Distributed Security Symposium*, 2011.

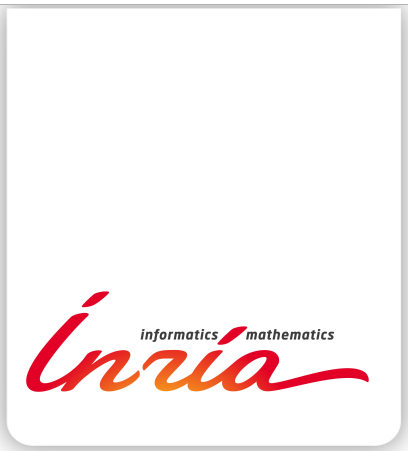
[22] Kenneth G. Paterson, Thomas Ristenpart, and Thomas Shrimpton. Tag size does matter: Attacks and proofs for the TLS record protocol. In *Advances in Cryptology - ASIACRYPT*, pages 372–389, 2011.

[23] E. Rescorla. HTTP over TLS, 2000. Request for Comments 2818, IETF.

[24] Qixiang Sun, Daniel R. Simon, Yi-Min Wang, Wilf Russell, Venkata N. Padmanabhan, and Lili Qiu. Statistical identification of encrypted web browsing traffic. In *IEEE Symposium on Security and Privacy*, pages 19–30, 2002.

[25] Serge Vaudenay. Security flaws induced by CBC padding - applications to SSL, IPSEC, WTLS ... In *Advances in Cryptology - EUROCRYPT*, pages 534–546, 2002.

[26] Charles V. Wright, Scott E. Coull, and Fabian Monrose. Traffic morphing: An efficient defense against statistical traffic analysis. In *Network and Distributed Security Symposium*, 2009.



RESEARCH CENTRE
PARIS – ROCQUENCOURT

Domaine de Voluceau, - Rocquencourt
B.P. 105 - 78153 Le Chesnay Cedex

Publisher
Inria
Domaine de Voluceau - Rocquencourt
BP 105 - 78153 Le Chesnay Cedex
inria.fr

ISSN 0249-6399